Pushing XML Queries inside Relational Databases

Ioana Manolescu, Daniela Florescu, Donald Kossmann

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Pushing XML Queries inside Relational Databases

Ioana Manolescu — Daniela Florescu — Donald Kossmann

N° 4112

January 2001

THÈME 3

ISSN 0249-6399 ISRN INRIA/RR--4112--FR+ENG

Pushing XML Queries inside Relational Databases

Ioana Manolescu * , Daniela Florescu † , Donald Kossmann ‡

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Caravel

Rapport de recherche n° 4112 — January 2001 — 41 pages

Abstract: Due to the increasing usage of XML as an exchange format, querying XML documents is also gaining attention. While several languages exist, work is under way to define the W3C standard for an XML query language. To take full advantage of the strength of the RDBMS, XML queries on XML virtual documents need to be translated to SQL queries to be executed by a relational engine. Several existing projects propose mappings between XML documents and relational databases, but the query translation problem from XML queries to SQL queries received less attention. We propose a general method for pushing XML queries into a relational database containing the XML document's data. Our solution is *general* in that it fits a wide range of mappings between the relational data and the XML documents; it makes a maximal use of the relational engine query capabilities, and relies on well-known algorithms from the relational database field, which makes it a good candidate solution for adding XML query capabilities to a relational system. We also investigate the limits of the translation process in general - not everything can be translated, given the big difference between the expressive power of SQL and that to be expected from XML query languages; we provide an analysis of language features to be expected from the standard XML query language, and establish which of them can be translated to SQL and how.

Key-words: XML, semistructured data, Quilt, query translation, query rewriting

* INRIA, Caravel project. Contact: Ioana.Manolescu@inria.fr
† INRIA, Caravel project. Contact: Daniela.Florescu@inria.fr
‡ Technical University of Munich, Germany. Contact: kossmann@informatik.tu-muenchen.de

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Execution des requêtes XML dans des systèmes de gestion de données relationnels

Résumé : Avec la popularité croissante de XML comme format d'échange de données, les langages de reques pour XML font aussi l'objet de nombreuses études. Tandis que plusieurs langages ont déjà été proposés, le W3C (World Wide Web Consortium) a mis en place un groupe de travail dont la tâche est de définir un nouveau standard pour un langage de requêtes XML. Pour le stockage des données XML, les systes de gestion des bases de données relationnels (SGBDR) sont une solution intéressante, à cause de leur fiabilité et de leurs performances. Pour profiter pleinement de la puissance d'un SGBDR, les requêtes XML sur des documents virtuels doivent être traduites dans des requêtes SQL à exécuter par le moteur relationnel. Plusieurs projets existants proposent des mappings entre des documents XML et des bases de données relationelles, mais le problème de la translation des requêtes XML vers SQL n'a pas été, à ce jour, bien étudié.
Nous proposons une méthode générale pour traduire des requêtes sur un document XML vers des requêtes SQL sur la base contenant les données qui composent ce document. Notre solution est *générale* par le fait qu'elle convient à un large spectre de mappings entre des données relationelles et des documents XML; cette méthode utilise au maximum les capacités de traitement de requêtes du systeme relationnel, ce qui en fait une bonne candidate pour ajouter des capabilités XML à un SGBDR existant. Nous nous intéressons aussi aux limites du processus de translation en général: toute re quête XML ne peut pas être traduite, à cause des differences significatives entre le pouvoir d'expression de SQL et le pouvoir d'expression probable du langage de requêtes pour XML; nous analysons les élé ments qui se trouveront, probablement, dans le futur standard de langage de requêtes XML, et nous montrons lesquels pourront être traduits en SQL, et comment.

Mots-clés : XML, données sémi-structurées, Quilt, translation de requêtes, réecriture de requêtes

1 Introduction

XML has become the standard for information interchange, due to its flexibility, portability and simplicity. Increasing XML adoption in the industry has fostered intense research and standardization efforts on a data model, query language, typing mechanism etc. specifically designed for XML. Query languages in particular received a good deal of attention, while storage systems and query processors for native XML data are currently in a more incipient phase.

Since most of the world's business data is stored in relational (or object-relational) format, it is quite likely that a large fraction of the XML data exchanged and queried within diverse organizations consists in fact of virtual XML documents whose data is physically stored in a relational DBMS. End users view data under the form of XML documents, and query it using some XML query language. At this point, two issues can be identified: how to map between virtual XML documents and relational tables storing the data; and how to translate XML queries on the virtual documents into SQL queries on the real data.

To solve the first problem, several mapping techniques exist, proposed by the database research community, or released in commercial products; the complementary half of such solutions is obviously the translation of XML queries to SQL. This paper focuses on *investigating whether and how can XML queries be translated into SQL.* We examine the feasibility of the translation, since there is a serious data model and query capability mismatch between SQL and an XML query language; and, when the translation is possible, we provide the proper translation rules.

Two questions arise at this point: which XML query language and which data mapping we consider. A standard query language for XML has not yet been defined, but the W3C body in charge of its ellaboration has issued (preliminary) versions of the query algebra [24], data model [28], and requirements [25] for the future standard language; these documents serve as the basis for our study. In the absence of a standardized syntax, we use the Quilt query language [3] for illustrating our examples.

As for the mapping, we believe there is no single "good" one, since different document types, data instances and query workloads may entail widely different mappings between relational data and XML views of this data. Therefore, we see the need for a *general* algorithm that translates XML queries on the XML views into SQL queries on the data stored in relational engines, *regardless of the data mapping.* We identify a significant subset of the possible mapping schemes, rich enough to contain the solutions previously proposed in research projects (or implemented in industrial products), and we provide a translation algorithm that applies to this family of mappings.

Our solution proceeds in three steps :

1. **normalizing** the incoming XML query by applying equivalence rules, bringing it to a form which is more appropriate for translation; this is described in section 5.

2. **translating** the normalized XML query into SQL on a *virtual, generic relational* schema, that can model any XML document; translation rules are presented in section 6.

3. **relational query rewriting** of the query that resulted from the translation: we use the relations in the physical storage as materialized views over the generic schema, as detailed in section 7.

This last query rewriting step determines the mapping set for which our solution holds: those mappings that allow the relational storage to be described as view over our virtual generic schema.

Physical data independence This three-levels approach has a big practical advantage: it separates the logical and physical views of the data, and decouples query translation from the identification of the tables needed to answer the XML queries. The logical-physical independence, following the idea from GMaps [22], introduces more degrees of freedom in linking relational and XML data; for example, if a new table containing XML information is added, the whole mapping does not need to change: we only need provide a definition of the table as a view over the generic schema. Forbidding access to a table is also easy - remove the corresponding view definition. Furthermore, a logical-physical separation allows seamless treatment of redundant relational storage: if redundant relational views are materialized, we will be able to use them exactly like the base tables, by exploiting their view definitions over the generic schema.

Novelty of our approach Our contributions can be outlined as follows. We provide equivalence rules for Quilt query rewriting; these rules allow us to normalize queries, for an easier translation to SQL. We analyze the features required from the standard query language, and explain how each of them can translate to SQL; also, we provide insights on specific constructs that cannot be processed by a single pass in a relational engine. We exploit semantic information about the virtual generic schema to prune useless views from the SQL query rewriting process, to speed up the rewriting. Finally, we propose a *general* framework for translating Quilt queries independently of the XML-relational data mapping, thus completely separating the storage design from the query translation.

We have implemented all of the described transformations (normalization, translation, rewriting) on top of a research relational database prototype; in general, the cummulated time spent in these steps is insignificant. This execution time is not a limitation for the global approach (virtual XML corresponding to real relational data); the existing limitations are explained in sections 5, 6 and 7, and arise from the mismatch between the SQL and XML data models.

In the following section, we give a formal description of the problem, survey related work, and outline our contribution. We then proceed to describe the current state of the algebra, data model, and query language requirements identified by the W3C group (section 3). We devote some space to a description of the Quilt query language in section 4, before describing the three query processing steps in our framework. We conclude in section 8.

2 Problem definition and overview of our approach

We place ourselves in the following context: we are given a relational engine and some view definition mechanism linking data in relational tables and virtual XML views of this data. In this context, we investigate *whether and how is it possible to execute XML queries on these virtual views within the relational DBMS.*

Up to now, the query translation problem has only been touched within the context of a given particular mapping technique. Before outlining our approach in section 2.2, we introduce a sample database for illustration, and we briefly describe existing proposals of mappings between XML and relational database in section 2.1.

Sample data For illustration, we use a sample database taken from the context of health professionals information systems, shown in figure 1. This is a likely field of application since related data collections have accumulated in RDBMSs for a long time, while a complete set of DTDs describing various data structures (e.g. for patient information, diseases, diagnosis, examinations) has recently been established [29]. Figure 1 shows sample tuples from two tables: one concerning patient information, and the other containing entries for several patients, in a global medical record. Each entry in the Record table has an unique ID *entryID*, and may point to a previous record that is relevant for the current record (e.g. previous examination linked to the same disease). Underneath the tuples, we inserted one possible way of representing the same data in XML. XML documents can be seen as ordered trees of elements. Elements have tags, and may have: nested element children as is the case for the *record* elements, simple text content like the *medication* elements, and attributes, as for example *entID*. Children elements are ordered within their parents, while attributes of an element are unordered.

In the most general case, an XML document is only required to be properly nested (well-formed); optionally, a type description (DTD) can be attached to a document, and we show DTDs for the sample XML documents in figure 1. A DTD can contrive the structure of a document by declaring what content is allowed within each element; it can also declare an XML attribute of special type ID: an ID uniquely identifies the element within a document, as is the case for *ssNo* and *entID*. Another special kind of attribute, IDREF, is used to "point to" element IDs; e.g. the *rel_previous* attribute links the owner element to the one having an ID attribute whose value is "1". For simplicity, we will only use DTDs as type description for the illustrating documents; we will mention later XML Schema, a more recent and powerful formalism for describing XML structures, that is gradually replacing DTDs.

2.1 Existing work on mapping between XML and relational

These mappings are divided in two cathegories, according to the sense of the mapping: from XML to relational or the opposite.

Relational storage schemas for XML documents XML-centric approaches were the first to come by, since they inherited naturally of the efforts toward storing semistructured

Patient(name:String, dob:Date, **SSno**:Long, address:String) ("Doe,John", "1/1/60", 123, <"1, South St., Palm Beach, FL") ("Ale, Mary", "2/6/70", 101, "2, Pine Rd., Bear Canyon, MN")	Record(**entryID**:Long, SSno:Long, date:Date, symptoms:String,diagnosis:String, medication: String, rel_previous:Long) (1, 123, "1/9/90", "fatigue, bad sleep", null, "blood tests", null) (2, 123, "10/9/90", "low blood iron", "Anemy", "Biofer", 1)
<patient> <tuple SSno="123"> <name>"Doe,John"</name> <dob>"1/1/1960"</dob> <address>"1, South St., Palm Beach, FL"</> </tuple> </tuple> <tuple SSno="101"> <name>"Ale, Mary"</name> <dob>"2/6/1970"</dob> <address>"2,Pine Rd., Bear Canyon, MN"</> </tuple> </patient> <!ELEMENT patient (tuple*)> <!ELEMENT tuple (name dob address)> <!ATTLIST tuple SSno ID #REQUIRED> <!ELEMENT name #PCDATA> <!ELEMENT dob #PCDATA> <!ELEMENT address #PCDATA>	<records> <record><patientSSno>"123"</> <entry entID="1"> <date>"1/9/90"</> <symptoms>"fatigue, bad sleep"</> <diagnosis></> <medication>"blood tests"</> </entry> <entry entID="2" rel_previous="1"> <date>"10/9/90"</> <symptoms>"low blood iron"</> <diagnosis>"Anemy"</> <medication>"Biofer once a day"</> </entry> </record> </records> <!ELEMENT records (record*)> <!ELEMENT record (entry+) patientSSno> <!ELEMENT patientSSno #PCDATA> <!ELEMENT entry (date symptoms diagnosis medication)> <!ATTLIST entry entID ID #REQUIRED, rel_previous IDREF> <!ELEMENT date #PCDATA> <!ELEMENT symptoms #PCDATA> <!ELEMENT diagnosis #PCDATA> <!ELEMENT medication #PCDATA>

Figure 1: Relational tables containing medical data and sample DTDs.

data in relational storage systems. In a nutshell, the goal is to find good relational storage schemas for XML documents, according to some criteria (e.g. number of tables, data instance and query mix, number of nulls etc.)

The Stored [7] project uses a data mining algorithm for identifying regular structures in a collection of XML documents without DTDs. For each "stable" pattern found in the data, Stored allocates one table; the authors propose to store the data items that do not belong to any stable pattern in a semistructured storage, while we consider that all the XML data we intend to query is stored within the RDBMS. Whatever the storage for the overflow, exploring it to answer queries entails a serious performance hit, since there is no schema for that data; the author's hypothesis is that the overflow graph is "often" small. For each mapping that the data mining algorithm discovers, translation rules are generated for pushing XML queries to the RDBMS. We note that the query language used does not construct new structure, it can only retrieve existing elements; also, to ensure losslessness of mappings, the mapping language is restricted (e.g. no joins): our method applies to a wider range of mappings, including those that join in a single table information from two different documents. Also, we do not forbid "lossy" mappings - there are cases when the XML queries are known in advance and they do not need all the data. In our framework, if the relational views do not contain enough information, this is discovered in the last relational rewriting phase described in section 7. I

The study in [10] models XML documents as ordered, oriented graphs, and investigates several ways of storing first, links between nodes, and second, flat values. For storing link information, the simplest storage scheme materializes a single relation for all graph edges; a more refined approach partitions the edge table on the edge label. For storing values, a simple solution is to have one table per value type; such type partitioning is beneficial for domain-specific indexing. A second approach for storing values is to add in the edge tables one column per type of value possible; the destination field of every edge is either a node or a value of a given type. This paper does not provide algorithms for query translation. Performance measures on a relational RDBMS for several types of XML queries show that the partitioned edge approach, combined with inlining values, offer the best performance. This is understandable since (a) self-joins on the complete Edge table are avoided and (b) joins to retrieve values are unnecesary if they are nested within elements.

Finally, [16] discusses three mappings derived from a simplified graph structure of the DTDs. Nodes in the graph are constructed for each element type; a parent-child or ID-IDREF relationship in the DTD is modeled by a directed edge from the parent node to the child node, therefore such graphs may have cycles. Three mappings are proposed, ranging between fully normalized and highly redundant; the interest of the mappings is assessed by counting the number of joins required to reconstruct the document. We note that this measure does not account for dataset particularities or a query mix. The mappings that provides the best performance, called *hybrid*, allocates a table for one element type as soon as its corresponding node in the graph (a) has zero in-degree or (b) it belongs to at least a loop in the graph, and: either has in-degree greater than one, or can appear any number of times ("*") in a parent. All elements that do not meet these criteria are stored within

the closest ancestor for which a table was created. The authors sketch the translation from XML-QL queries on the *shared* mapping to SQL; our method generalizes their results for a larger family of mappings, and provides a more thorough investigation on translating features of a full-fledged XML query language.

Defining XML views of relational tables More recently, the mapping problem has been tackled with a table-centric plan of attack: given existing relational data collections, the focus is on how to efficiently export and query them under the form of XML documents.

XPERANTO [2] proposes a *default* XML Schema for mapping any relational table; one example is shown in the medical database in figure 1, for the Patient table. This simple mapping reflects the table/tuple/attribute structure of the table; we show a DTD for brevity. More complex mappings, like the one for the Records table in figure 1 (that has grouped record entries according to the SSno of the patient), can be defined as XML queries against the default mapping. User queries against custom views are composed with the view definitions and result in XML queries against the default mapping. A graph representation of these queries is very close to the internal SQL query representation in the IBM DB2 commercial product, and enables the translation of the XML query to a SQL query; no further translation details are provided.

In [17], the authors are concerned with alternative ways of efficiently constructing fully materialized XML views of the data. The mapping language is SQL extended with element constructors; that study is not concerned with XML query translation to SQL.

Finally, the SilkRoute project [8] uses a special language RXL for specifying XML views of relational data. An RXL query has a "retrieve" part that gathers data from relational tables using an SQL-like syntax, and a "construct" part that builds XML structure using the relational data. To group output information items into a single element, the "construct" part of an RXL query describes them as children of an element whose uniqueness is guaranteed by the query execution unit. This can be accomplished by assigning to an element an ID attribute whose value is computer by a Skolem function, as it was done, for example, in XML-QL[6]. User XML queries are composed with RXL view definitions and result in new RXL, that in turn is translated to SQL queries sent to the execution engine; the fact that the mapping is defined from relational to XML determines the very nature of RXL and of the translation process. In contrast, we propose a more general framework designed to deal with both relational-to-XML and XML-to-relational mappings; also, our translation method does not need any other language beside an XML query language and SQL.

2.2 Translating XML queries in the general case

We hope by now the reader is convinced that a great variety of mappings can be imagined between XML document and relational tables storing their data. The source of this variety resides in the data instance, the application or storage constraints, the most frequent queries, and the sense of the translation (whether it goes from XML to relational or the opposite).

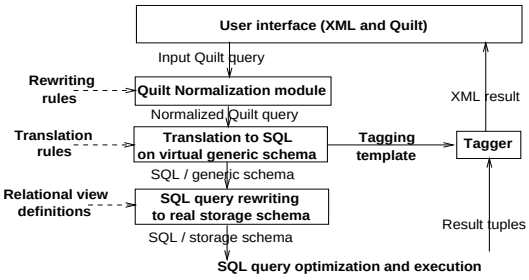

Figure 2: General architecture.

From our perspective, imposing a specific mapping technique is not at all desirable; particular applications are entitled to their particular mappings, just as the view selection problem in the relational database world has long been identified as depending on the data and application characteristics (see, e.g., [21]). A frequent scenario of XML usage leads to the same conclusion: DTDs or Schemas are established for a particular application domain, and from then on, actors in the field begin exchanging their (relational) data mapped to the standardized format. It is highly unlikely that a single mapping might suit them all; even more so since their data collections have private, proprietary schemas that will not be shown, and even less modified to fit a new XML structure !

Therefore, our goal is to find a methodology for translating XML queries on XML documents into SQL queries on the equivalent relational data *regardless of the mapping between the two*; and to do this by exploiting as much as possible first, relational databases themselves, for the execution, and in a more general view, the existing body of research and experience in the field of relational databases. We attempt, whenever possible, to make use of algorithms already studied and implemented in existing industrial products; this should provide for a painless enhancement of existing DBMSs with XML query capabilities.

The architecture we propose, presented in figure 2, consists of the following layers:

- XML normalization module : this module, described in section 5, applies rewriting rules to bring the user query to one of the forms that we are able to translate to SQL.
- Translator from Quilt queries to SQL: this module incrementally applies the rules given in section 6, that translate Quilt constructs into SQL over a virtual relational schema described in section 2.3.
- SQL query rewriter: using definitions of data as views over the virtual schema, the rewriter outputs a query that can be executed by the relational engine. We show in section 7 what modifications we suggest for simple relational rewriting algorithms to deal with the kinds of queries that we produce.
- Tagging module: this module is in charge of the construction of new elements that constitute the output of the query. To describe the structure of the result, a tagging

Document(**docID**, docURIID, rootElemID)	Element(**elID**, elQNameID, elTypeID)
URI(**uriID**, uriValID)	NameSpace(**nsID**, nsValID, nsURIID)
ProcInstr(**piID**, piVal1ID, piVal2ID)	Comment(**commID**, commValID)
QName(**qNameID**, qnPrefixID, qnLocalID)	Value(**valID**, value)
Attribute(**attrID**, attrElID, attrNameID, attrValID)	Child(**parentID**, childID, childValID, **index**)
TransClosure(parentID,childID)	

Figure 3: Virtual Generic Schema for XML Documents

template is computed during the translation from Quilt to SQL and is passed directly from the translator to the tagger. We describe briefly the construction of the template and the functioning of the tagger in section 6.2; this component is inspired by work done in [17].

2.3 Virtual generic schema as support for translation

The simple generic, virtual, relational schema that we use is shown in figure 3. This schema is constructed as a fully normalized relational version of the hierarchical structure of an XML document; note the use of foreign keys (shown in bold) to represent the filiation of different entities within a document. The last table, TransClosure, is redundant with respect to the rest of the schema; it represents the pairs of elements that are linked by an ancestor-descendent relationship (i.e. the transitive closure of the parent-child relationship represented by the Child table). The TransClosure table is useful for translating recursive XML path expressions; we will show in section 7 how the rewriting algorithm deals with it.

Using the virtual generic schema has several advantages. First, it separates the logical and physical views of the data, therefore it frees us from having to provide a separate translation algorithm per mapping. Second, by being so close to the structure of a document, it also offers good support for query translation, as we show in section 4. Third, examining the translation process on such a simple and generic structure, we can assess which XML query language constructs are feasible within a RDBMS *independently of the mapping scheme*, and which are not.

Document order and persistent element IDs The virtual generic schema reflects the order of children within their parent in the index field of the Child table; but global order of XML elements (also called *document order*) is not explicitly represented. If the data comes from an RDBMS (i.e. we are defining an XML view of an existing relational database), element order is not really a concern - there is no order of tuples within a table; if there is an order among the data items modeled, it is materialized in a tuple field, and as such, our translation scheme can handle it. If the order was meaningful (e.g. the original format was XML and document order does matter), the elID field of the Element table can be made to reflect the index of the element in the document. A problem still remains - it is not enough

to represent order in the virtual schema, the storage must also reflect it in order to be able to query it, e.g. by adding an elID field reflecting the document order whenever an element is stored. We stress the fact that this is a general issue to be solved *independently of the mapping and query translation* used, not a limitation of our three-steps approach: in all the surveyed related projects, the same approach is taken. The implicit document order found in XML needs to be explicitly represented as a data order in an RDBMS, or it will be lost.

3 XML query data model and query algebra

Research on semistructured data Even if XML is barely two years old, the database community had, since 1995, investigated the *O*bject *E*xchange *M*odel [14] for semistructured data. OEM data instances are schemaless labeled unordered trees; the absence of order is the main difference between OEM and XML. The interesting aspect of OEM was the absence of any *a priori* type specification or constraint on the data; but this freedom proved too expensive to pay, since repositories and query processors for such data were very difficult to build, and OEM had no direct impact on the industry.

W3C standards More recently, industrial application started using XML, and with XML, a set of related standards for manipulating XML documents. These standards, issued by the W3C, come with their *implicit* data model, in the sense that they give some description of what a document is, from their target applications point of view. To ensure interoperability among W3C specifications, features of several related standards are taken into consideration during the standardization process of an XML query language; we briefly describe these features.

XML Infoset [23] describes an abstract data set containing the information to be found in a well-formed XML document, e.g. element and attribute filiation, attributes names and values. XML Schema [26] is a formalism for describing data structures within XML documents. XML Schema provides a very rich type system: many atomic types, union types, inheritance by derivation, typed tuples (sequences), typed lists, complex nested structures etc.

XPath [27] was designed as a language for addressing parts of an XML document. An XPath expression consists of a sequence of navigation steps; at every step, a current node list named *context list* stores the current state of the evaluation. Applying a navigation step to a node in the context list yields a new list of nodes. The new context list resulting from the one-step navigation is the concatenation of the lists obtained for each node in the old context list. Let us consider an example: we evaluate the XPath expression *//entry/medication* in the "record.xml" document shown in figure 1. Since "//" means "all descendents starting from the document root", the first navigation step retrieves all the *entry* elements and inserts them into the new context list c_1. The second navigation step retrieves all *medication* children of the two entries, and concatenates the result into a single list c_2; the order within c_2 is dictated first, by the order of the *entry* elements, and next, by the order of the *medication* children within their parents.

Note on XPath XPath has a very simple type system: it contains nodes, node lists, and a few atomic types like numeric and string. Since this standard was released before XML Schema, a path expression navigating within an XML document for which there is an XML Schema can be typed in two different ways: following the XPath type system, or following the XML Schema specification.

In XPath, several powerful type casts are applied automatically. The most important feature is *list flattening*: XPath only operates with lists and unnests lists automatically; this means, e.g., that $[1,[[2,3],4]]$ is always reduced to $[1,2,3,4]$. This flattening is also retained in the W3C working draft for an algebra, and it marks an important difference w/r to the way relational and object-oriented algebras work; as we will show in section 5, the implicit flattening leads to new query rewriting rules, while rules which used to apply in SQL and OQL [20] do no longer work.

There are also differences between the casts that XPath performs and those retained in the algebra; the algebra is much more strict with typing. For example, XPath provides cast rules for comparing a list of elements and a boolean value (using a hidden existential quantifier, and downcasting elements to strings); such a comparison is a type error in the algebra. Also, XPath only compares nodes by their string value, while in the algebra there is also an identity-based comparison etc. While semantic differences exist between XPath and the query language algebra, XPath is an established standard (and a constituent part of other popular standards such as XSLT); therefore, a simple subset of XPath is quite likely to be used for specifying path expressions in the standard XML query language, and this is also the case in Quilt.

The W3C data model and query algebra The most significant *mandatory requirements for the data model* are that it must show how all its constructs can be expressed only in terms of the InfoSet, and it must represent complex types as described in the XML Schema. The *data model type system* has a comprehensive collection of simple atomic types; *list, set, bag* and *tuple* constructors allow for composition of more ellaborate types, as well as *union* types. In a manner similar to DOM and XPath, a generic *Node* type is provided, which is the union of types *Document, Element, Attribute, Value, NameSpace* and such. From an element, the data model provides accessors to its children, attributes, parent, and type; the type of any XML element is an XML element itself. Among the *mandatory requirements for the query language*, we cite:

declarativeness;	closure with respect to the data model;
querying documents with or without a schema;	possibility to combine several documents;
support for namespaces;	support for null values;
support for querying hierarchy and sequence;	preservation of hierarchy and sequence;
sorting operator;	preservation of node identity;
existential and universal quantifier;	simple operations on names;
creation of new document structures;	reference traversals, dereferencing operator;
composition of operations (and of queries);	support for aggregation

Non-mandatory requirements for the language include: support for external functions, integration of the XPath syntax, and access to the DTD or XSchema of a document. For each of the mandatory features, in section 4 we show the corresponding Quilt syntax, and in section 6 we discuss whether and how it can be translated to SQL. We refer to the features listed in the query requirements and not the query algebra for two reasons. First, the requirements document is much more stable; the algebra is still under discussion; second, requirements are more general and permit a more high-level analysis of language features. With respect to the algebra, we mention, however, that it provides a strong typing system, and that it also features implicit list flattening.

4 The Quilt query language

This language is a recent proposal for an XML query language and combines the benefits and useful features of several languages already known in the literature, such as [20],[6], [27] and [1]. Quilt is a functional language specifically designed for XML. A central notion in Quilt is that of *expression*; starting from constants and variables, expressions can be nested and combined, using the usual arithmetic, logical and set operators, navigation primitives, function calls, higher order operators like sort, conditional expressions, element constructors etc.

Path Expressions In this respect, Quilt borrows the abbreviated syntax of XPath; we illustrate path expressions with examples. The *document("mySite.org/records.xml")* expression retrieves the root of the XML document situated at the given URI. The Quilt path expression *document("records.xml")/record* is the ordered list of all *record* children of the document root. The expression *document("records.xml")//record* designates the list of *record* elements at any depth in the document, in document order. In Quilt, a path expression can start in a document root or in an implicit current root node that is deduced from the evaluation context; the expression *//entry/@ssNo* retrieves the collection of values of the ssNo attributes in all *entry* elements in the current document. A dereference operator is also provided: *//entry/@rel_previous→entry* returns all medical entries that are "pointed at" by some other entry descendant of the current node. Specifying the tag of the target element is not mandatory ("*" can be provided instead of the tag).

Path expression can contain path predicates that restrict the navigation to nodes satisfying certain logical conditions. For example, *//entry[date = "1/9/90"]* has the following meaning: all the *entry* elements having at least one *date* child, whose string value is "1/9/90", are returned. A path predicate can also select a specific range; as an example, d*ocument("records.xml")//entry[range 2 to 5]* will only return the second to fifth *entry* elements, in document order. From this description of Quilt path expressions, it can be seen that automatic list flattening applies here, too. With respect to the requirements, Quilt path expressions provide for hierarchy and sequence querying and preservation, and for reference traversal.

```
for     $r in documents("records.xml")//record,
        $e in $r/entry
where   $e/date > "1/1/90" and
        contains($e/diagnosis, "pollution")
return  <polutionIncident>
            $r/@ssNo,
            $e/diagnosis
        </polutionIncident>
```

(a)

```
for     $no in distinct(documents("records.xml")//record/@ssNo
        let $recs:=documents("records.xm")//record[@ssNo=$no]
return  <pollutionIncident>$no,
            (for $e in $recs
            where $e/date > "1/1/90" and
                contains($e/diagnosis, "pollution")
            return $e/diagnosis)
        </pollutionIncident>
```

(b)

Figure 4: Sample Quilt queries.

Operators Quilt provides the usual set of first-order operators (arithmetic, logical and set-oriented); the comma is a list concatenation operator. For example, *(//entry, //name)* returns the concatenation of the list of all entries, followed by all names, in document order. Second order operators in Quilt are the logical quantifiers *any* and *all*, and *sort*. For example *document("records.xml")//entry sort by date desc* will return all *entry* elements, the most recent first.

Conditional expressions These expressions correspond directly to an important feature of the algebra. The syntax is *if* C *then* E_1 *else* E_2; an important thing to note is that the type of this expression is the union of the types resulting from the two branches. As we will see in section 6.3, this typing feature has some consequences on query translation.

Element constructors This class of expressions provides for construction of new XML structure; for example, *<alphalist>(document("patient.xml")//name) asc </alphalist>* constructs an alphabetic list of all patient names. Syntactically, the element's tag, attribute names and attribute values can be obtained from constants or variables, while the children are specified as a list of arbitrary expressions.

FLWR expressions These expressions consists of three parts: a *for-let* clause, that make variables iterate over the result of an expression, or binds variables to arbitrary expressions, a *where* clause that allows the specification of restrictions on the variables, and a *return* clause, that can construct new XML elements as output of the query. For example, the query in figure 4(a) retrieves all the medical records of people with health problems that have been related to pollution within the last ten years. Variables defined with a *for* are iterators: *$r* variable iterates over all the *record* elements, and *$e* over the entries in the record associated to *$r*. For each *($r, $e)* pair that satisfies the condition, a new *pollutionIncident* element is created, containing the patient SSno and the diagnosis. A variant in figure 4(b) *groups* the interesting entries according to the patient's SSno; the variable *$recs* is defined with a *let* clause, which means *$rec* is bound to the whole value of the expression assigned to it (as opposed to *$no* which iterates over the expression in the *for* clause). The *return* clause contains a nested FLWR expression, that is dependent on the externally bound variable *$no*.

FLWR expressions allow combining data from different documents, grouping, construction of new structure, and are natural candidates for query composition.

Functions Quilt provides a syntax for defining functions; these can be called in Quilt queries. In particular, recursive queries can only be implemented in Quilt by calls to recursive functions. The syntax for a function definition is illustrated in the following example; we show the function declaration and a sample query calling it. The function pastEntries returns the list of all past record entries linked to a given record. This example also features a function from its standard library, *empty* (other functions like *count*, *max*, *avg* etc. are included).

```
function   pastEntries(ELEMENT $e) RETURNS [ELEMENT]
           { if empty ($e/@rel_previous →entry)
           then []
           else ($e/@rel_previous→entry UNION pastEntries($e/@rel_previous→entry)}

for        $e in document("records.xml")//record[@ssNo="123]/entry
where      $e/date="1/1/1990"
return     pastEntries($e)
```

General form of a query We can now describe the general structure of a query in Quilt: it consists of an optional list of namespace definitions, followed by a list of function definitions, followed by a single *expression*. It is important to notice that in Quilt there is no "first-class citizen" data type - *document("patient.xml")//entry*, as well as *(5+1)* and *avg([2, 3])* are legal Quilt queries; also, by its functional nature, it allows arbitrary nesting of expressions within a query. Nested queries tend to be more difficult to translate because SQL does not allow the same freedom in nesting; this is why in the following section we provide rules for unnesting Quilt queries. Redundant Quilt constructs are also eliminated in this phase.

5 Quilt query normalization

Having introduced the language and the data model, we proceed now to describe the first step in our three-steps architecture: the Quilt query normalization phase, during which we reduce queries to simpler forms, more appropriate for query translation. Also, at this level, we can already identify query structures for which it is impossible to push all but the tagging step in the relational query engine, and explain why.

Traditionally, rewriting rules are formulated in terms of an algebra; space limitations forbid the introduction of the formalism and notations set for the algebra; since language constructs are simple enough to reason directly at this level, our rewriting rules are shown in a bare Quilt syntax.

We use the following notations: x, y, z correspond to individual Quilt query variables, capital letters like E, R, C denote Quilt expressions. For brevity, we sometimes write a single for clause "for $\vec{x}$ in E" instead of "for x_1 in E_1, x_2 in $E_2(x_1)$, ... x_n in $E_n(x_1, \ldots x_{n-1})$"; in this case, E is an expression of arity n, and $\vec{x}$ are consecutively bound to each tuple of values that result from E's evaluation.

Elimination of temporary variables The first rewriting rule that we give is used to eliminate temporary variable definitions. *Let* clauses, like the one in figure 4(b) are just syntactic sugar, and as such, they are eliminated during the normalization: the expression binding the variable (in the rule, $E_2(x)$) is simply substituted to all occurences of the variable y. As a general rule, whenever it can be inferred (e.g. by using a DTD) that an iterator variable introduced with *for x in E* can only have one value, we replace its declaration with *let x := E* and then treat it like any other *let*.

RR_1	for $\vec{x}$ in E_1 let $y = E_2(\vec{x})$ for $\vec{z}$ in $E_3(\vec{x}, y)$ where $C(\vec{x}, y, \vec{z})$ return $R(\vec{x}, y, \vec{z})$	$\Rightarrow$	for $\vec{x}$ in E_1, $\vec{z}$ in $E_3(\vec{x}, E_2(\vec{x}))$ where $C(\vec{x}, E_2(\vec{x}), \vec{z})$ return $R(\vec{x}, E_2(\vec{x}), \vec{z})$

Expressions built on top of FLWR expressions Any kind of expression can be built on top of a FLWR expression; we show at left the general form of such expressions. In the particular case when E_1 is a sequence of "/" or "//" steps, eventually interspersed with path predicates, but without the *range* predicate, the rewriting rule RR_2 applies. The nametest can be a string constant, variable, or "*".

E_1(for $\vec{x}$ in E_2, where $C(\vec{x})$ return $E_3(\vec{x})$)	RR_2	(for $\vec{x}$ in E_2, where $C(\vec{x})$ return $E_3(\vec{x})$) $/nameTest$	$\Rightarrow$	for $\vec{x}$ in E_2, where $C(\vec{x})$ return $E_3(\vec{x}, y)/nameTest$

The intuition behind this rule is that both FLWR expressions and path expressions (without *range*) commute with the list constructor; this rule is one of the consequences of the automatic list flattening. The *range* predicate does not have this property: *(for $x in //record return $x/entry) [range 2 to 5]* returns at most three *entry* elements, while *for $x in //record return $x/entry[2 to 5]* may return up to three entries of each record; this is why RR_2 does not hold for *range* predicates. Element constructors are another unnestable case; indeed, *<res>for $x in //record return $x </res>* constructs a single *<res>* element, while *for $x in //record return <res>$x</res>* constructs a new element per record found.

Expressions built on top of element constructors We aim at translating as much as possible of a query in SQL - and only construct some XML at the end, as shown in figure 2;

a relational query processing engine cannot deal with intermediate XML results. Therefore, we attempt to unnest such expressions whenever possible; if we cannot unnest, then the query cannot be translated to SQL, nor executed in a relational engine. This category of queries can only be handled by adding the possibility to model and store intermediate XML documents as such; in this paper, we aim at describing exactly what can be done only as an extra layer on top of an existing RDBMS.

The general form of such expressions is $E(EC(\vec{x}))$, where $\vec{x}$ represent variables that may have been bound outside this expression; we identify three cases.

1. In some restricted cases when E consists of path steps without the *range* predicate, then they can be pushed inside the element constructor, making it disappear. This is similar to the elimination of useless element constructors accomplished during RXL view composition in [8]; we provide the simplification rule RR_3 as a rule among others, that apply together to normalize the XML query itself. The application range of the rule is discussed in more details below, next to RR_3.

2. Some other expressions that depend on an XML element can be evaluated without actually computing the element; instead, the expression can be evaluated on the element's children and the results are aggregated to obtain the correct result on the element. For example, consider the *width* function, that computes the width of an element viewed as a tree of nodes; we can tell that by summing the widths of all its children, we obtain the width of the parent. To take advantage of such cases, the rewriter has to be aware of the particular semantic of such functions.

3. If E uses an operator that cannot be "pushed" inside an element constructor, then EC cannot be suppressed. The most frequent case is when E calls an external function for which we have no semantic information. As explained, such cases cannot be handled by a SQL processing chain.

Rule RR_3 shows how to push *//nameTest* steps into element constructors, in the case where all E_is are of element type; if some flat values are interspersed with the element children, they are erased by the translation. The same rule holds for attribute steps.

RR_3	(<tag>		
	$E_1(\vec{x})$		$(E_1(\vec{x})$//nameTest
	...	$\Rightarrow$	...
	$E_n(\vec{x})$		$E_n(\vec{x})$ //nameTest)
	</tag>)//nameTest		

Note that this rule does no longer hold for */nameTest* steps; there are several differences. First, if for some i, $E_i = F/nameTest$, then *(<tag>*E_i*</tag>)/nameTest* = *(<tag>*$F/nameTest$*</tag>)/nameTest* = F. This entails that the exact value of *nameTest*, as well as the tags of the elements from each E_i have to be known statically, before running the query, in order to decide what is the correct result. Obviously, if one of them is computed dynamically, no rewriting applies.

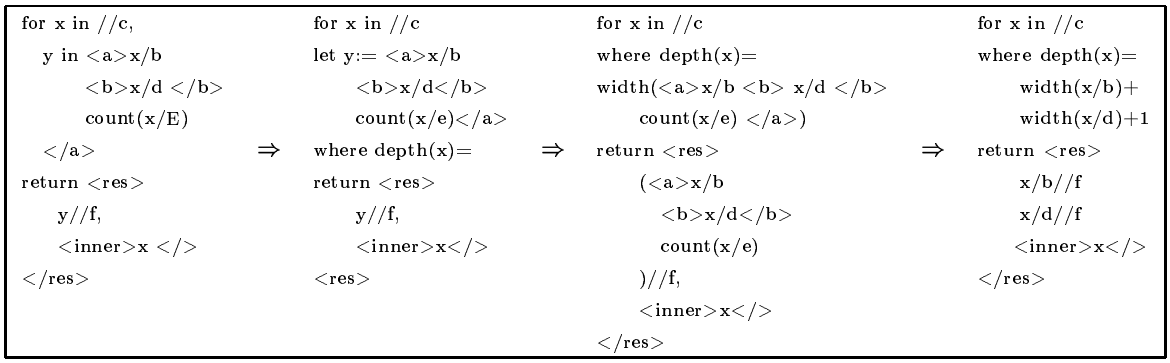

Figure 5: Sample rewriting of for an element constructor in the for clause: replacement with local variable definition, suppression of the local variable, semantic decomposition.

Element constructors nested in FLWR expressions The key observation in these cases is that an element constructor of the form $EC(\vec{x})$ creates exactly one element per tuple of variables it depends on. The following rule shows how to introduce a *let* declaration, and from now on we can apply R_1; to know if this query can ever be translated to SQL, we need to apply to E_2, C and R the same discussion as in the previous case.

RR_4	for $\vec{x}$ in E_1		for $\vec{x}$ in E_1
	y in $EC(\vec{x})$		let y := $EC(\vec{x})$
	for $\vec{z}$ in $E_2(\vec{x}, y)$	$\Rightarrow$	for $\vec{z}$ in $E_2(\vec{x}, y)$
	where $C(\vec{x}, y, \vec{z})$		where $C(\vec{x}, y, \vec{z})$
	return $R(\vec{x}, y, \vec{z})$		return $R(\vec{x}, y, \vec{z})$

An element constructor appearing directly in a *where* clause is a type error, since it cannot evaluate to a boolean, which is expected in such places. An element constructor may appear within a *where* clause as a subexpression of something more complex; what can be done depends again on the exact expression, as described when discussing RR_2.

Several levels of nested element constructors are a normal component of *return* clauses; we translate them directly to SQL as shown in section 6.2. If opaque functions are applied to subelements of the constructed result, the query is not translatable to SQL, for the same reason: an all-SQL framework cannot materialize intermediate XML results.

Figure 5 shows an example of applying RR_4 followed by RR_1 and RR_3. Following the notations in RR_4, E_1 is *//c*, there are no z variables depending on the constructed element, C is *depth(x)=width(y)*, and R is the *<res>...</res>* expression. A nice side effect is that *count(x/e)* is erased from the query and x/e never computed, since it is not needed in the result.

Nested FLWR expressions in a for clause If the inner FLWR expression returns newly constructed elements, the discussion is the same as for nested element constructors. Otherwise, the simple rewriting rule RR_5 unnests the inner FLWR expression.The reason behind this rule is again the implicit list flattening feature of the algebra; such a rule does not hold in OQL.

RR_5	for $\vec{x}$ in E_1, y in (for $\vec{z}$ in $E_2(\vec{x})$ where $C_1(\vec{x},\vec{z})$ return $E_3(\vec{x},\vec{z})$) where $C_2(\vec{x},y)$ return $E_4(\vec{x},y)$	$\Rightarrow$	for $\vec{x}$ in E_1, $\vec{z}$ in $E_2(\vec{x})$, y in $E_3(\vec{x},\vec{z})$ where $C_1(\vec{x},\vec{z})$ and $C_2(\vec{x},y)$ return $E_4(\vec{x},y)$

For each binding of $\vec{x}$, there is a list of associated $\vec{z}$ bindings, and for each pair of $(\vec{x},\vec{z})$ bindings, there is a list of bindings for y. The rule basically says that the levels of list nesting do not matter, i.e. y iterates over the same list of bindings, whether they come in small sublists, corresponding to each $\vec{z}$ binding in the original query, or in a single list, that results from the triple iteration on the right.

Nested FLWR expressions in a where clause In the following rule, the inner FLWR expression must return exactly one (boolean) value per binding of $\vec{x}$, otherwise the query is ill-typed; this means that $E_2(\vec{x})$ also produces a single value. Therefore, it is as if $\vec{z}$ had been introduced with a *let* declaration; and *for-where-return* on a single value can be replaced by *if-then-else*, with an empty result on the *else* branch. In our case, since the expression needs to evaluate to a boolean, we write "else false". Eliminating the local variable declaration leads to the final result.

RR_6	for $\vec{x}$ in E_1 where (for $\vec{z}$ in $E_2(\vec{x})$ where $C_1(\vec{x},\vec{z})$ return $E_3(\vec{x},\vec{z})$) return $E_4(\vec{x})$	$\Rightarrow$	for $\vec{x}$ in E_1 where (let $\vec{z} := E_2(\vec{x})$ in (if $C_1(\vec{x},\vec{z})$ then $E_3(\vec{x},\vec{z})$ else $false$) return $E_4(\vec{x})$	$\Rightarrow$
$\Rightarrow$	for $\vec{x}$ in E_1 where (if $C_1(\vec{x},E_2(\vec{x}))$ then $E_3(\vec{x},E_2(\vec{x}))$ else $false$) return $E_4(\vec{x})$	$\Rightarrow$	where $C_1(\vec{x},E_2(\vec{x}))$ and $E_3(\vec{x},E_2(\vec{x}))$ return $E_4(\vec{x})$	

If the inner FLWR expression appears inside a more complex one, the unnesting (and the question whether the query can be processed through SQL or not) depend on the form of the enclosing expression; the discussion surrounding RR_2 applies.

Nested FLWR expressions in a return clause Rule RR_7 applies to FLWR expressions appearing directly in a return clause; if FLWR expressions appear nested within a more complicated construct, the unnesting and the translation depend on the form of that expression. We illustrate the rule with an example:

RR_7	for $\vec{x}$ in E_1 where $C_1(\vec{x})$ return (for $\vec{y}$ in $E_2(\vec{x})$ where $C_2(\vec{x}, \vec{y})$ return $E_3(\vec{x}, \vec{y})$)	$\Rightarrow$	for $\vec{x}$ in E_1, z in (for $\vec{y}$ in $E_2(x)$ where $C_2(\vec{x}, \vec{y})$ return $E_3(\vec{x}, \vec{y})$) where $C_1(\vec{x})$ return z

	for x in document("records.xml")//entry
	where x/date="1/9/90"
	return
	(for y in documents("patient.xml")//records
	where y/@ssNo=x/SSno
	return y)
		$\Rightarrow$	for x in document("records.xml")//entry,
			z in (for y in documents("patient.xml")//records
			where y/@ssNo=x/SSno
			return y)
			where x/date="1/9/90"
			return z

Eliminating conditional expressions Rule RR_5 shows how to deal with conditional expressions. RR_8(a) is meant for cases when E is constructed only with path expression steps (including path predicates with *range* tests), element constructors, or arbitrary function calls (user-defined or from the standard library). RR_8(b) to (d) show how to eliminate path expressions when they are directly nested within a *where*, *return* or *for* clause; note that rules (c) and (d) modify the order of the result, therefore an extra sorting step would be needed on the result of the union.

RR_8 (a)	E(if $C(x)$ then $E_1(x)$ else $E_2(x)$)	$\Rightarrow$	if $C(x)$ then $E(E_1(x))$ else $E(E_2(x))$
(b)	for $\vec{x}$ in E_1 where (if $C_1(\vec{x})$ then $E_2(\vec{x})$ else $E_3(\vec{x})$) return $E_4(\vec{x})$	$\Rightarrow$	for $\vec{x}$ in E_1 where ($C_1(\vec{x})$ and $E_2(\vec{x})$) or ($\neg C_1(\vec{x})$ and $E_3(\vec{x})$) return $E_4(\vec{x})$
(c)	for $\vec{x}$ in E_1 where $C_1(\vec{x})$ return (if $C_2(\vec{x})$ then $E_2(\vec{x})$ else $E_3(\vec{x})$)	$\Rightarrow$	for $\vec{x}$ in E_1 where $C_1(\vec{x})$ and $C_2(\vec{x})$ return $E_2(\vec{x})$ $\cup$ for $\vec{x}$ in E_1 where $\neg C_1(\vec{x})$ and $C_2(\vec{x})$ return $E_3(\vec{x})$
(d)	for $\vec{x}$ in E_1 y in (if $C_1(\vec{x})$ then $E_2(\vec{x})$ else $E_3(\vec{x})$) where $C_2(\vec{x}, y)$ return $E_4(\vec{x}, y)$	$\Rightarrow$	for $\vec{x}$ in E_1, y in $E_2(\vec{x})$ where $C_1(\vec{x})$ and $C_2(\vec{x}, y)$ return $E_4(\vec{x}, y)$ $\cup$ for $\vec{x}$ in E_1, y in $E_3(\vec{x})$ where $\neg C_1(\vec{x})$ and $C_2(\vec{x}, y)$ return $E_4(\vec{x}, y)$

Nested conditional expressions of the form *if* C_1 *then (if* C_2 *then* E_2 *else* E_3*) else* E_4 can be easily unnested; for brevity we do not show the translation rules, which are trivial. We only note that every level of unnesting doubles the number of union terms.

Transforming path predicates in where clauses Path predicates (without *range*) appearing in a *for* clause inside a FLWR expressions can be eliminated and moved to the *where* clause, since both have existential semantics. In rule RR_9, the right hand side makes explicit the dependency of E_2 and E_3 on the newly introduced variable y; this dependency was implied by the position of the path predicate in the query on the left. We illustrate the rule with an example.

RR_9	for x in $E_1[E_2\ op\ E_3]/PE$ where $C_1(x)$ return $E_4(x)$	$\Rightarrow$	for y in E_1, x in y/PE where $C_1(x)$ and ($E_2(y)$ op $E_3(y)$) return $E_4(x)$
	for x in document("records.xml")// record[SSno="123"]/entry return x/medication;	$\Rightarrow$	for y in document("records.xml")//record, x in y/entry where y/SSno="123" return x/medication;

6 Translating from Quilt to SQL

We now turn to explaining the translation between the normalized Quilt query and the SQL query on the generic schema. This schema represents in some sense "the least common denominator" for a wide category of mappings between XML and relational, enabling us to focus on the simple translation of language features, show how these can be achieved, and pinpoint particular difficulties that can be clearly attributed to the language mismatch and not to lossy mappings.

Typing the result of the query As explained in section 3, the XML query language algebra provides strict typing rules, type inference is done on the query before it is handled to the translator. Therefore, when translating an expression, we know whether the result of the translation is, from the SQL point of view, a value, row or table query. If a relational engine obtains several rows from a subquery that was supposed to return a single value, a runtime error occurs; several XML query languages implicitly casted collections to singletons. This used to be a big difference between XML query languages and SQL; strict typing restricts the XML query language and facilitates the translation.

6.1 Translating path expressions

Let us denote by $T(E) = (S(E), F(E), W(E))$ the translation function that, for a given expression E, computes the select, from and where parts of the corresponding SQL query, denoted by $S(E), F(E)$ and $W(E)$ respectively.

The first rule we give is the one that translates the path expression denoting a document root. Two cases are possible: the document name can be a string constant, or can result from a different query, but in any case, it needs to evaluate to one string value:

TR_1(a)	T(document($docName$))=	select d.docID from Document d, URI u, Value v where d.docURIID=u.uriID and u.uriValId=v.valID and v.value=$docName$
(b)	T(document(E))=	select d.docID from Document d, URI u, Value v where d.docURIID=u.uriID and u.uriValId=v.valID and v.value=($T(E)$)

The following rules show how to translate path expressions given the translation of the path shorter by one step. TR_2 shows how to add a final "child" step to the SQL translation of an expression; again, there are two slightly different cases, according to the name test being a constant or resulting from a complex expression. We show the rule for the most general case, when the name of the child results from a complex expression; if E_2 is a constant, simply replace $T(E_2)$ with the constant. Since the path expression is correctly typed, we

know that $S(E_1)$ must be an element ID, and that $T(E_2)$ must return a single row with one string column:

TR_2	$T(E_1/E_2) =$	select e.elID from $F(E_1)$, Child c, Element e, Qname q, Value v where $W(E_1)$ and c.parentID=$S(E_1)$ and c.childID=e.elID and e.elQNameID=q.qNameID and q.qnValID=v.valID and v.value=$T(E_2)$

Note that TR_2 outputs the ID of the child element; if in a certain situation, the *textual representation* of the element is desired, this information is inserted during the typing proces. In such a case, the translator receives an expression of the form $toText(E_1/E_2)$, where the *toText* function takes as argument an element ID and returns its text image. For such simple functions, the translator simply copies the function into SQL, i.e. $T(toText(E_1/E_2)) = toText(T(E_1/E_2))$.

We move on to translate the expressions whose final step is a "descendant" step, denoted by "//". Note the use of the TransClosure table to express arbitrary depth nesting; as before, $T(E_2)$ should be replaced with the constant name of the descendent, if it is statically known.

TR_3	$T(E_1//E_2) =$	select e.elID from $F(E_1)$, TransClosure tc, Element e, QName q, Value v where $W(E_1)$ and $S(E_1)$=tc.parentID and tc.childID=e.elID and e.elQNameID=q.qNameID and q.qnLocalID=v.valID and v.value=$T(E_2)$

The next rule shows how to translate a final "attribute" step; the rule has two variants as the previous ones, according to whether the attribute name is a string constant or results from a different expression. In this rule also, $attName$ is an expression which returns a single string; if the attribute name is a constant, that constant replaces $T(attName)$.

TR_4	$T(E_1/@attName) =$	select a.attrID from $F(E_1)$, Attribute a, Value v where $W(E_1)$ and a.attrElID=$S(E_1)$ and a.attrNameID=v.valID and v.value=$T(attName)$

Rule TR_5 translates a dereferencing step. The only subtlety is that the name of the ID attribute in the target element needs to be inserted, to ensure a correct matching between the ID and IDREF values; the user does not need to know this name, but in the presence of the DTD, the query translator will insert it (without a DTD, the dereference operator does not make sense). We only show the case when the attribute name is a constant; the example following the rule illustrates rules TR_1, TR_3, and TR_5.

TR_5	$T(E_1/@attName \rightarrow elName) =$ ID of $elName$ is attribute id	select e.elID from $F(E_1)$, Attribute a1, Value v1, Value v2, Element e, QName q, Value v3, Attribute a2, Value v4, Value v5 where $W(E_1)$ and a1.attrElID=$S(E_1)$ and a1.attrNameID=v1.valID and v1.value=$attName$ and a1.attrValID=v2.valID and e.elQNameID=q.qNameID and q.qnLocalID=v3.valID and v3.value=$elName$ and a2.attrElID=e.elID and a2.attrNameID=v4.valID and v4.value=id and a2.attrValID=v5.valID and v2.value=v5.value

T(document("records.xml")//entry/ @rel_previous→entry)=	select e2.elID from Document d, URI u, Value v0, TransClosure tc, Element e1, QName q1, Value v1, Attribute a1, Value v2, Element e2, QName q2, Value v3, Attribute a2, Value v4, Value v6 where d.docURIID=u.uriID and u.uriValID=v0.valID and v0.value="records.xml" and d.rootElemID=tc.parentID and tc.childID=e1.elID and e1.elQNameID=q1.qNameID and q1.qnLocalID=v1.valID and v1.value="entry" and e1.elID=a.attrElID and a.attrNameID=v2.valID and v2.value="rel_previous" and a.attrValID=v3.valID and e2.elQNameID=q2.qNameID and q2.qnLocalID=v3.valID and v3.value="entry" and a2.attrElID=e2.elID and a2.attrNameID=v4.valID and v4.value="entryID" and a2.attrValID=v5.value and v3.value=v5.value

Unnesting subqueries into joins The reader may have noticed that when discussing rules TR_1-TR_5, in the case where the child, descendent or attribute name came from a different expression, we inserted a nested query corresponding to the translation of that expression. Since the nested query is guaranteed to return one value (i.e. is a row query returning one string) for every parent element, it could be easily unnested into a join. For illustration, we wrote the translation example above as unnested.

Correctly ordering the result In general, the results of path expressions should come in document order; SQL queries, however, do not guarantee result order, unless an explicit sort step is added. Remember from the discussion in section 2.3 that we require element IDs to reflect document order. To correctly order the translation results, one only needs to add, for example, to $T(E_1/E_2)$, "sort by $S(E_1)$, e.elID". Even if $T(E_1)$ was already sorted on $S(E_1)$, after the extra joins the ordering needs to be re-established.

Translating the *range* predicate We deliberately omitted it until now, since it is one of the features that are rather difficult to translate. Remember that E*[range* n_1 *to* n_2*]* returns the items whose ordinal position in E's result is between n_1 and n_2. The feasibility of the translation depends on the nature of the order implied in the result of E.

If it is a *data order,* i.e. it corresponds to an order of data items materialized in a table, then *range* can be translated easily in SQL: we show here the translation of a generic path expression, where we suppose that $T(E)$ results in a table with a single column, named $elID$. This SQL query selects the elements within the correct range in document order (the order resulting from $elID$); we ask for tuples such that the tuples preceding them are no less than n and at most p.

TR_6	$T(E[range\ n\ to\ p]) =$	with V as $(T(E))$
		select $x.elID$ from V x
		where (select count(*) from V y where $x.elID > y.elID) \geq n$
		and
		(select count(*) from V y where $x.elID > y.elID) < p$

This rule can be easily extended to cases when the order results from a cross-product, as in the case in FLWR expressions (described in section 6.2). We omit the details to avoid clutter; we only mention that the comparison $x.elID > y.elID$ needs to be replaced with the proper comparisons for establishing that $(x_1, x_2, \ldots, x_n) > (y_1, y_2, \ldots, y_n)$ in the sense of the desired lexicographic order.

If the order with respect to which we want to perform a *range* is *ad-hoc*, i.e. cannot be deduced by comparing stored values, the range predicate cannot be translated in a single SQL query. Consider an example: *(f(//entry))[range 2 to 5],* where we assume that f takes a list of elements and returns the same list whose order was scrambled somehow. Intuitively, we cannot translate such predicates because we have no sort key on which to compare the items in the result of *f*, to decide which items go first.

6.2 Translating FLWR expressions

We recall from section 4 that the *for* clause produces tuples of bindings for the variables in the query, the *where* clause poses extra conditions that discard some of these tuples, and the *return* clause uses the tuples of bindings that satisfy the selection conditions to construct the result, either under the form of complex structured XML elements or as tuples of flat values.

Let us first consider a simple FLWR expression where all expressions in the *for* and *where* clauses are path expressions, and that can only return all the variables bound in *for-where* (these may be quite useless queries, but we will move later to more interesting ones).

TR_7	T(for x_1 in E_1, x_2 in $E_2(x_1)$, ... x_n in $E_n(x_1, \ldots x_{n-1})$ where $C(x_1, \ldots x_n)$ return $x_1, \ldots x_n$) =	select $S(E_1), S(E_2) \ldots S(E_n)$ from $F(E_1), \ldots F(E_n)$ where $W(E_1)$ and ... and $W(E_n)$ and $T(C(x_1, \ldots x_n)$
	T(for $x1 in document("records.xml")//entry, $x2 in $x1/date where $x2="1/9/90" return $x1, $x2) =	select e1.elID, e2.elID from Document d, URI u, Value v1, TransClosure tc, Element e1, QName q1, Value v2, Child c1, Element e2, QName q2, Value v3, Child c2, Value v4 where d.docURIID=u.uriID and u.uriValID=v1.valID and v1.value="records.xml" and d.rootElemID=tc.parentID and d.rootElemID=tc.parentID and tc.childID=e1.elID and e1.elQNameID=q1.qNameID and q1.qnLocalID=v2.valID and v2.value="entry" and c1.parentID=e1.elID and c1.childID=e2.elID and e2.elQNameID=q2.qnLocalID and q2.qnLocalID=v3.valID and v3.value="date" and e2.elID=c2.parentID and c2.childValID=v4.valID and v4.value="1/9/90"

Iterations and result ordering To respect the semantic of XML, tuples returned by path expressions should have the order that is the lexicographic order derived from the order in each E_i. From a database point of view, ignoring the order would result in more efficient execution plans. Depending on the application domain, if the ordering of tuples is important, a final *sort by* x_1 *asc, ...,* x_n *asc* needs to be added.

Much more interesting are queries that return new XML elements, by restructuring content from the queried documents; restructuring and element construction are among the mandatory requirements for an XML query language. To explain the translation of such queries, let us first show how to translate element constructors; in general, an element constructor depends on variables bound in the *for-where* clause.

Translating element constructors We introduced the syntax for element constructors in section 4. To correctly structure and order the information needed in the construction of an element, we borrow the *sorted outer union* approach presented in [17]. We formalize the translation as follows.

Let E_0 be the part of the query providing bindings for the query variables $\overrightarrow{x} = x_1, \ldots, x_n$ (in the case of FLWR expressions, the *for-where* clauses); every tuple resulting from $T(E_0)$ contains bindings for the variables in $\overrightarrow{x}$. The tag of the outermost result element might depend on the bound variables $\overrightarrow{x}$; we denote the tag by $E_1(\overrightarrow{x})$. Let $E_2, \ldots, E_{2k}$ be the expressions providing names for the element's attributes, while $E_3, \ldots, E_{2k+1}$ provide attribute values. Let $H_1, \ldots, H_j$ be the expressions corresponding to the result element's content. Finally, let $G_1, \ldots, G_l$ be the elementary expressions (no element constructor) appearing in

TR_8	$T(< E_1(x) \quad E_2(\vec{x}) = E_3(\vec{x}) \ldots$	with $T(E_0)$ as BoundVars
	$\ldots E_{2k}(\vec{x}) = E_{2k+1}(\vec{x}) >$	(select bv.*, 0 as label, null as $g_1, \ldots,$ null as g_l
	$H_1(\vec{x})$	from BoundVars bv
	$\ldots$	$\cup$
	$H_j(\vec{x})$	select bv.*, 1 as label, $S(G_1)$ as $g_1, \ldots,$ null as g_l
	$</tag>)=$	from BoundVars bv, $F(G_1)$
		where $W(G_1)$
		$\cup \ldots$
		select bv.*, $l+1$ as label, null as $g_1, \ldots, S(G_l)$ as g_l
		from BoundVars bv, $F(G_l)$
		where $W(G_l)$)
		sort by bv.*, label

Figure 6: Translation rule for element constructor.

the E_is and H_is, that really depend on the bound variables $\vec{x}$; each G_i provides values to be used as attribute or element names, attribute values, or character data. The rule is shown in figure 6. The first union term corresponds exactly to the *for-where* clause, padded with nulls; this term contains only the variable bindings, and is labeled 0. Each of the next l terms retrieves the information corresponding to one of the $G_1, \ldots, G_l$ path expressions. The label is used by the tagger module, as we explain further.

The tagging template The sorted union query lines up in good order tuples containing all the necessary *data* to build the new element; *structure* is not represented, and this is the role of the tagging template mentioned in section 2.2. Figure 7 shows an example: the Quilt query assembles information from the two medical data sources (the join is performed in the where clause). On our sample database, there is only one binding for the variables $x_1, x_2, k = 0$ (no attributes in the returned element), $j = 2$, H_1 is the element constructor with tag *personal*, H_2 is the element constructor with tag *medical*; $l = 3$, G_1 is *$x_1/name*, G_2 is *$x_1/address*, G_3 is *$x_2/entry*.

The tagging template informs the tagger how to use the result set to construct output elements; this template is constructed during the translation of the FLWR expression. First, the *for-where* block is translated: this is $T(E_0)$ in figure 6 (x_1, x_2 in figure 7). The first union term of the query can now be partially constructed; after all the columns from BoundVars, we insert the label column, and we create a new template, in which we mark the column where the label is stored. Next, we copy the structure of the returned element into the template as follows. Whenever an attribute's name or value or a subelement's tag or content correspond to constants, they are copied in the template. Whenever we encounter a G_i, we add an union term to the sorted union query, joining the result of the *for-where* block and the translation of G_i; also, we fill in the template with the name of the result column where values for G_i

```
for $x1 in document("patient.xml")//tuple,
  $x2 in document("records.xml")//record
where $x1/@SSno=x2/patientSSno
return <medFile>
  <personal><patName> $x1/name </>
    <patAddress> $x1/address </></>
  <medical> $x2/entry </medical>
</medFile>
```

x_1	x_2	label	g_1	g_2	g_3
t_2	r_1	0	-	-	-
t_2	r_1	2	"Doe..."	-	-
t_2	r_1	3	-	"1, S St..."	-
t_2	r_1	4	-	-	<entry eID="1">...</>
t_2	r_1	4	-	-	<entry eID="2">...</>

```
<template disc="label">
<elem tag="medFile">
  <elem tag="personal">
    <elem tag="patName">
      <directContent col="g1"/>
    </elem>
    <elem tag="patAddress">
      <directContent col="g2"/>
    </elem>
  </elem>
  <elem tag="medical">
    <directContent col="g3"/>
  </elem>
</elem></template>
```

```
<medFile>
  <personal>
    <patName>"Doe,John"</patName>
    <patAddress>"1, South Street, Palm Beach, FL"</patAddress>
  </personal>
  <medical>
    <entry eID="1"><date>"1/9/90"</date>
      <symptoms>"fatigue,bad sleep"</>
      <medications>"blood tests"</></entry>
    <entry eID="2"><date>"10/9/90"</date>
      <symptoms>"Anemy (low blood iron)"</>
      <medications>"Biofer"</><rel_previous>"1"</></entry>
  </medical>
</medfile>
```

Figure 7: Translation of a return clause containing element constructors: original query, result of the sorted outer union, tagging template, and final result.

can be found. This amounts to multiple outer joins between the bound variables and the expressions retrieving components of the result, that depend on these variables[1].

Each block of the sorted union query will be rewritten separately (the rewriting algorithm is discussed in section 7) and handled to the execution engine; the result metadata (column number, types and names) stay the same in the queries over the virtual and real schemas. The tagger receives a stream of real data, and uses the tagging template to construct the XML result: for every line labeled 0, it starts a new element with the correct tag; then, by following the labels of result tuples, it decides where to fill in the value from the single g_i column that is not null. The tagger functions in linear time and constant space; for more details, we refer the reader to [17].

[1] Recently, an optimization algorithm for SQL queries with joins and outer joins has been presented in [15].

6.3 Translating heterogeneous type queries

The reason why this translation is not straightforward is that the algebra allows heterogeneous unions, of union types; for example, a query might return the union between a set of strings and a set of integers. There are two ways around it: either we are willing to cast one or both results to bring them to the same type (complex XML types end up as flattened values anyway), or we produce one column in the result per different type and fill in nulls. In the later case, the tagger will choose alternatively a value from one column or another, whichever is not null; this information is inserted in the tagging template. We omit the details.

6.4 Translating functions and function calls

The XML Query requirements identify three classes of functions: the language's standard library of functions (e.g. *count*, *distinct*, *document*), functions that the user may define using the query language and use in queries, and external functions, for which a "black-box" implementation and a typed signature is provided (support for these functions is not mandatory).

Functions in the standard library Among the standard functions of an XML query language, some correspond to functions or operators available in SQL, like *count* and *distinct*. Others, such as *document*, have a special meaning w/r to the virtual schema, and therefore translate to small SQL queries. Of course, the query translator must be aware of the semantic of special functions from the standard library.

User-defined functions The definitions of such functions, which are in principle instances of the XML query language itself, cannot be translated to SQL in the general case - that is, if they actually use their arguments - since SQL does not provide for parameterized views. As far as the translation of the function call is concerned, two cases may arise. If the function is not recursive, then its definition is merely a macro; the function call can be replaced in the query with the function body, applying the proper substitutions. If the function is recursive, then there may be an equivalent SQL query (making use of recursive view definitions) that has the same effect. However, note that the latest SQL standard provides only for linear, stratified recursion [18, 9]; anything more powerful cannot have a (standard) SQL equivalent.

Besides the classical SQL that we use here, the standard [18] provides for procedural *P*ersistent *S*tored *M*odules (PSM), embedded SQL, and client APIs as alternative means of exploiting the system's processing engine. Since most current RDBMSs implement at least some of these features, pretty much everything can be accomplished at this level, if we are willing to move functions explicitly defined in the XML query language to the category of externally implemented functions.

6.5 Other language constructs

We group here several language features whose translation is easy to do, but for space reasons we can only hint at how it is done; we then discuss more difficult translations, first some for which a satisfactory solution can be found in our setting, and finally some that cannot be pushed into relational engines, independently of the mapping and query translation mechanism that is used.

The following XML query features can be pushed easily the relational engine:

Logical operations Negation is a very general feature that will probably be allowed in the standard language. However, special care needs to be taken in translating to SQL since in XPath (and probably in the standard query language too), several predicates have a hidden existential semantics. Also, logical connectors (*and, or*) do not pose problems.

Quantifiers Among the mandatory requirements for an XML query language are universal and existential quantifiers; these can be translated easily using the corresponding SQL *exists* construct.

Text operations An important subset of these operations consists of text search predicates; SQL/MM provides a rich extension to SQL to deal with text search queries and it is worth investigating the integration between these features and those that will be retained in the XML query language standard.

Translating the following operations needs some extra care; the discussion is valid in general, whatever method is used for mapping and query translation.

Sorting The *sort* operator of Quilt can in general be translated to the corresponding SQL operator, but a difference needs to be resolved. In Quilt, sort can apply on any expression, not necessarily the result of a query; in SQL, sort can only apply to a cursor, i.e. at the top level in a query (not in a view, not in an innner query block). Therefore, sort at top-level is easily supported, but order in subqueries is difficult to do; in some cases, it can be simulated by a final sorting step.

Identity-based operations In Quilt, set operations (intersection, set difference) are based on identity of items; in SQL, such operators are rather based on values; also, there are contexts when equality is understood as identity of the objects compared. The virtual schema provides conceptually unique IDs for all data items in a document; for the complete structure to work, the virtual IDs should have some real conterpart, i.e. stored IDs need to be present in the real data to make the translation mechanism work. As explained in section 2.3, persistent IDs are required also for reconstructing document order.

XML-specific functions. Here we refer to the class of functions providing the string image of various XML entities, e.g. the full text contained within an element, or

the concatenation of its text descendents. These functions could be implemented as external functions, and may be generated automatically from the mapping structure. In any case, the mapping information is hardwired inside such functions, since they need to be aware of where to find the data contained in each type of element.

We now draw the list of features whose translation from Quilt to SQL is rather problematic, *independently of the mapping mechanism* used, and for whom we see no satisfactory solution.

Constant expressions cannot be translated to SQL, even if they are legal Quilt queries. The SQL standard [18] does provide a *values()* row and table constructor, but for the minimal conformance to the standard, *values()* is only required in insert statements, and several major RDBMSs only supply it in these contexts.

Metadata queries For both relational and XML, metadata is modeled by data objects: system catalogs in an RDBMS are themselves tables, while element types correspond to actual XML elements according to the XML query data model [28]. In the case of XML, type can be queried in the same query as data; the data model provides an accessor function that steps from an element to its corresponding type. The expressive power of SQL does not allow queries that mix data and metadata (this can only be done in more powerful extension called SchemaSQL [12]). Therefore, relational data and metadata cannot be queried in a single step. The only possibility is to query the mapping itself for the link between relational and XML metadata, and then query the right relational data; but these manipulations cannot be done in SQL itself, they are only supported at an external level (e.g. Java/JDBC). In any case, such queries, natural in Quilt, entail an exponential number of corresponding SQL queries (in the size of the relational catalogs).

Runtime access to type The query algebra provides operators that check at runtime if a data item is of an exact type (or a subtype of a given type). In its full generality, this cannot be done in SQL, since at this level we only manipulate flat values.

7 Query Rewriting

Until now, we have shown how to normalize Quilt queries, and how to translate them in SQL over the virtual schema, when the translation is possible. In this section, we detail how to obtain a SQL query on real data collections: we rewrite the query on the virtual schema using the real tables as views.

Equivalent rewriting for bag semantics When rewriting our queries, we use an algorithm that searches for equivalent rewritings only. It might be argued that in a data integration context, contained rewritings (partial answers) are more appropriate; in our setting, however, a query returns all the data that qualifies, since this is the general underlying assumption when using an RDBMS. Moreover, we are rewriting SQL on the virtual schema

to SQL on the real schema, and SQL has bag semantics; for this reason, and also to respect the logic of Quilt that by default preserves dupplicates, we are interested in rewriting algorithms for bag semantics. In the next section, we describe the mechanism of view definitions; section 7.2 details the algorithms that we use and the particular optimizations that apply for our queries.

7.1 View definitions

In order to actually query a relational table, there must be at least one view definition describing it as a SQL query over the virtual schema shown in figure 3. This condition determines *the family of mappings that our approach can handle*: it is exactly the set of mappings for which the relational sources can be described as views over the virtual generic schema. This is the case for the mappings proposed in [10] and [16], and for the default mapping in [2]. Since our virtual schema expresses the information contained in the XML InfoSet, and the standard query language should only rely on this information, all mappings defined as XML queries over a default XML mapping as in [2], or as mixed SQL-XML queries over relations as in [8], are within the expressive power of our mapping. The mappings constructed in Stored are also definable in terms of our virtual schema. We note that [16], [7] and [8] all use languages that rely on Skolem functions for grouping; Skolem functions are not part of the query algebra, and grouping can be accomplished by element constructors and nested iterations, as demonstrated in section 4.

As an example, figure 8 shows the view definition describing the Patient table from figure 1 as a view over the virtual schema. While quite large, this view is in fact very simple; it relates the information in the table to data items from the "patient.xml" document. The first three tables in the from clause, and the first three predicates in the where, give the name of the document. The next few joins represent the information that the root element of the document, e1, has a *patient* tag, while the joins in line 11 of the view retrieve its *tuple* children. For each *tuple* element (e2 in the query), the *SSno* attribute of the element provides the SSno field in the Patient table (v5.value is the actual value to be found in the element). Lines 14-16, 17-19 and 20-22 have the same structure; they describe the *name*, *dob* and respectively *address* children of the tuple elements. Each of these three children elements (e3, e4 and e5) contains a value corresponding to a field in the Patient table's tuples; these values, v5, v7 and v9, appear in the project list. Note that this view, besides the actual attribute values, also exports element IDs of all elements in the view definition. We have already discussed the need for IDs in real data collections in section 2.3.

Defining the Record table as a view on the "records.xml" documents is similar to the previous one, and for space reasons we do not include it. We end by noting that in general, such a view can involve several documents, and possibly the same document more than once; also, a view can concern elements "from any document" (when the URI of the document is not specified).

```
select v7.value as name, v9.value as dob, v5.value as SSno, v11.value as address,                1
    e1.elID as e1, e2.elID as e2, e3.elID as e3, e4.elID as e4, e5.elID as e5, e6.elID as e6      2
from Document d1, URI u1, Value v1, Element e1, QName q1, Value v2, Child c1, Element e2,        3
    QName q2, Value v3,                                                                          4
    Attribute a1, Value v4, Value v5, Child c2, Element e3, QName q3, Value v6, Child c3,        5
    Value v7, Child c4,Element e4, QName q4, Value v8, Child c5, Value v9, Child c6              6
     Element e5, QName q5, Value v10, Child c7, Value v11                                        7
where d1.docURIID=u1.uriID and u1.uriValID=v1.valID and v1.value="patient.xml" and               8
    d1.rootElemID=e1.elID and e1.elQNameID=q1.qNameID and q1.qnLocalID=v2.valID and               9
     v2.value="patient" and c1.parentID=e1.elID and c1.childID=e2.elID and                       10
     e2.elQNameID=q2.qNameID and q2.qnLocalID=v3.valID and v3.value="tuple" and                  11
    a1.attrElID=e2.elID and a1.attrNameID=v4.valID and v4.value="SSno" and                       12
    a1.attrValID=v5.valID and                                                                    13
    c2.parentID=e2.elID and c2.childID=e3.elID and e3.elQNameID=q3.qNameID and                    14
    q3.qnLocalID=v6.valID and v6.value="name" and c3.parentID=e3.elID and                        15
    c3.childValID=v7.valID and                                                                   16
    c4.parentID=e2.elIDand c4.childID=e4.elID and e4.elQNameID=q4.qNameID and                    17
     q4.qnLocalID=v8.valID and v8.value="dob" and c5.parentID=e4.elID and                        18
    c5.childValID=v9.valID                                                                       19
    c6.parentID=e2.elID and c6.childID=e5.elID and e5.elQNameID=q5.qNameID and                    20
    q5.qnLocalID=v10.valID and v10.value="address" and c7.parentID=e5.elID and                   21
     c7.childValID=v11.valID                                                                     22
```

Figure 8: View definition for the Patient table

7.2 Rewriting algorithm

Although abundant work on query rewriting using views for set semantics exists, there are quite few results in the case of bag semantics. The result stated in [5] is that two conjunctive SQL queries are equivalent only if they are identical up to renaming and reordering of tables; this problem can be brought to the problem of deciding graph isomorphism, and is in NP. This forms the basis of our algorithm; we first explain all the necessary enhancements by the mean of an example, on a simple unnested SQL query.

Basic algorithm A simple algorithm resulting from [5] is the following. Given the SQL unnested, non-recursive[2] query Q and views $V_1, \ldots, V_n$, find all subsets $\{V_{i1}, V_{i2}, \ldots, V_{ik}\}$ of $V_1, \ldots, V_n$ such that Q is isomorphic to a $\sigma - \pi$ query over $V_{i1} \bowtie \ldots \bowtie V_{ik}$ (note that these subsets may contain repetitions).

Pruning the views by using document information Let us first consider the case when the XML documents referred to by a Quilt query are named (well-determined); for example, the following query joins data from exactly two documents, "patient.xml" and "records.xml".

```
for     $x in document("patient.xml")/patient/tuple,
        $y in document("records.xml")//SSno
where   $x/@SSno=$y
return  <res>$x/name, $y</res>
```

By looking at the Quilt query, this information is easily deducted; but such knowledge would be very useful at the relational query rewriting stage, and here is why.

When rewriting the query, if we had just one view corresponding to the "patient.xml" document, there would be little hope of finding a rewriting, in the absence of record information. Also, a relational view combining information from the "patient.xml" document and from an "insurance.xml" document is, in general, useless for rewriting our sample query; this is because the joins contained in the view, linking data from the two documents, may alter cardinalities of items from the "patient.xml" document. Finally, a view containing information only from the "insurance.xml" document cannot provide useful information for our query, since we assume that documents are unique (and naturally, different documents have disjoint contents, since it follows from the InfoSet and query data model that each element belongs exactly to one document etc.). The conclusion that we attain is the following: a view set $\{V_{i1}, V_{i2}, \ldots, V_{ik}\}$ may provide an equivalent rewriting of the query Q only if the bag of documents to which the views make reference is the same as the bag of documents contained in Q. If we denote by $db(q)$ the document bag of a SQL query q over the virtual schema, the usability condition for the view set reads: $db(V_{i1}) \cup db(V_{i2}) \cup \ldots \cup db(V_{ik}) = db(Q)$. This

[2] We do not consider select-project-join queries involving the TransClosure table to be recursive (at the syntactic level, they are not).

means an important pruning of the available views when rewriting a query, which is highly desirable.

Of course, we need to show how can this information be inferred from the *translation* of the Quilt query. Consider the translation of the last Quilt query:

select	e1.elID as $x, e2.elID as $y	p,r
from	Document d1, URI u1, Value v1, Element e1, QName q1, Value v2, Child c1, Element e2,	p
	QName q2, Value v3, Attribute a1, Value v4, Value v5,	p
	Document d2, URI u2, Value v6, TransClosure tc1, Element e3, QName q3,	r
	Value v7, Child c2, Value v8	r
where	d1.docURIID=u1.uriID and u1.uriValID=v1.valID and v1.value="patient.xml" and	p
	d1.rootElemId=e1.elID and e1.elQNameID=q1.qNameID and q1.qnLocalID=v2.valID and	p
	v2.value="patient" and c1.parentID=e1.elID and c1.childID=e2.elID and	p
	e2.elQameID=q2.qNameID and q2.qnLocalID=v3.valID and v3.value="tuple" and	p
	a1.attrElID=e2.elID and a2.attrNameID=v4.valID and v4.value="SSno" and	p
	a1.attrValID=v5.valID and a1.attrValID=v5.valID and d2.docURIID=u2.uriID and	p
	u2.uriValID=v6.valID and v6.value="records.xml" and d2.rootElemId=tc2.parentID and	r
	tc2.childID=e3.elID and e3.elQameID=q3.qNameID and q3.qnLocalID=v7.valueID and	r
	v7.value="SSno" and c2.parentID=e3.elID and c2.childValID=v8.valID and	r
	v5.value=v8.value	p,r

We have marked on the right of every line by a "p" or an "r" whether the attributes, tables and predicates in this line only concern the "patient.xml" document, or the "records.xml" document, or both. The partition is easy to do: by following the predicates in the *where* clause, one can infer that element e1 belongs to document d1, element e2 is a child of e1 and therefore belongs to the same document, as well as its attribute a1 etc. This query is therefore separated in two zones, z_1 corresponding to the "patient.xml" document, and z_2 to "records.xml". In general, we can write $Q = z_1 \bowtie z_2 \bowtie \ldots \bowtie z_m$, where z_i is a subquery corresponding to one document. In the above example, the join predicate between the two zones appears on the last line in the translated query.

The translation algorithm can now be outlined as follows:

```
    REWRITE(Q, V_1, ... V_n)
1   determine db(Q), z_is such that Q = z_1 ⋈ z_2 ⋈ ... ⋈ z_m; z_i refers to document doc(z_i)
2   foreach view set U = V_i1, ..., V_ik such that db(V_i1) ∪ db(V_i2) ∪ ... ∪ db(V_ik) = db(Q)
3     PR ← {ε} (empty rewriting)
4     foreach view V ∈ U
5       foreach {z_j1, ..., z_jl} ⊆ {z_1, ..., z_m} such that db(V) = {doc(z_j1), ..., doc(z_jl)}
6         compute R_V = {r | r partial rewriting of the form z_j1 ⋈ ... ⋈ z_jl = V}
7         PR' ← PR ∪ {s ⋈ r | s ∈ PR, v ∈ R_V}; PR ← PR'
8       end
9     end
10    foreach rewriting r ∈ PR
11      if Q can be written as πσ(r) then output solution πσ(r)
12    end
13  end
```

Line 6 in the algorithm searches for equivalent rewritings of Q subqueries, using exactly one view. In lines 1 and 6, the join predicates between z_is are simply those that connect them in the query - when they are absent, the join should be read as a cross-product. In line 7, the join predicate on r and s is the conjunction of all predicates that connect $\{z_{j1}, \ldots, z_{jl}\}$ and s.

On our sample database, there are only two views: V_1 describes the patient table as a view over the "patient.xml" document, and V_2 (which we did not show) relates the Record table to the "records.xml" document. The query that we shown is divided in two zones corresponding to the two documents. There is an unique view set satisfying the partition condition, $U = \{V_1, V_2\}$, and in step 6 of the algorithm, each zone is rewritten using one of the views. In the general case, we expect to have a large number of views, and our test condition on view combinations significantly reduces the work during rewriting.

Views on undetermined documents The above optimization holds if the view definitions name the documents they use. Another type of views may contain "all the *entry* elements from the documents" (that is, all the documents present in the system). Such views need to be tried for all query zones, since we do not know a priori whether they can serve.

This kind of view pruning improves performance, but we still need to solve two issues to make sure step 6 in the above algorithm actually works.

Exploiting functional dependencies in the view definitions If the rewriting algorithm searches in step 6 for exact isomorphism between the view and some query zones, it will always fail. After partitioning our sample query, consider the rewriting of the zone z_1 referring to data from the "patient.xml" document:

z_1	select	e1.elID as $x
	from	Document d1, URI u1, Value v1, Element e1, QName q1, Value v2, Child c1, Element e2, QName q2, Value v3, Attribute a1, Value v4, Value v5
	where	d1.docURIID=u1.uriID and u1.uriValID=v1.valID and v1.value="patient.xml" and d1.rootElemId=e1.elD and e1.elQNameID=q1.qNameID and q1.qnLocalID=v2.valID and v2.value="patient" and c1.parentID=e1.elID and c1.childID=e2.elID and e2.elQNameID=q2.qNameID and q2.qnLocalID=v3.valID and v3.value="tuple" and a1.attrElID=e1.elID and a1.attrNameID=v4.valID and v4.value="SSno" and a1.attrValID=v5.valID

By examining $db(z_1)$, we know that only the Patient view V_1 shown in figure 8 may be used for rewriting z_1, since it has the same document bag. It is easy to see V_1 can be written as $z_1 \bowtie \ldots$, since the from and where clauses of z_1 are prefixes of the from and where clauses in V_1, and the columns returned by z_1 are among those exported by the V_1. In a general setting, this means that the view cannot be used for an equivalent rewriting of z_1, since the extra joins might have altered cardinalities.

Due to the special schema on which our queries are formulated, however, there is a solution: use the DTDs, and the semantic knowledge about our virtual schema. From the DTD of the "patient.xml" document, we can infer that every *tuple* element has exactly one *dob* child, and exactly one *address* child; therefore, the Element-Child-Element joins shown in lines 10, 12 and 14 in the view do not duplicate or erase tuples. Furthermore, every XML element has exactly one qualified name, and the local part of the qualified name corresponds to exactly one value; thus, the Element-QName-Value join sequences in lines 14-15, 17-18 and 20-21 of the view are also one-to-one. The same holds for the Child-Value joins on lines 16, 19 and 22, since there is one single content in the *dob*, *address*, and *name* elements.

If no DTD or Schema is available for the documents being queried, then the Patient view cannot be used for equivalent rewriting of z_1 with respect to bag semantics. Whatever the mapping and the query translation mechanism, a table storing "all *tuple* elements with at least one *dob*, *address*, and *name* child" cannot provide a complete answer to "retrieve all *tuple* elements". This brings us back to the observation made in [5], requiring perfect match between two queries for them to be equivalent; the virtual schema approach simply brings this problem back in the relational setting, where we can characterize it easily and reason about it using functional dependencies. We expect, however, that in most cases, DTDs or Schemas are available for the queried documents; this is most likely if the mapping is done from relational to XML, and in any case necessary as a help to formulating XML queries.

Treatment of transitive closure We have considered a rewriting when neither the query nor the view contained the TransClosure table. We now explain how to deal with queries corresponding to path expressions like *document("records.xml")//SSno* from our sample query.

1. If the query and the view are both defined using transitive closure, then simply searching for perfect view-query matches will find correct rewritings.

2. If the view is defined without transitive closure, but the query uses it, then DTDs should be used to decide whether the view materializes all the possible paths the query refers to. In the absence of a DTD, we cannot use the view for an equivalent rewriting; the same discussion as above applies.

3. If the view is defined with transitive closure, but the query asks for a specific path in the data, the feasibility of the query depends on the presence of a DTD and on the mapping. Consider a view storing *patientSSno* elements at any depth within *"records.xml"*, and a query asking for *document("records.xml")/records/record/patientSSno*. Using a DTD, the rewriter might infer that all *patientSSno* elements are found only on the path shown in the query. Without the path information present in a DTD, the view may contain a superset of the required data, and therefore cannot be used.

We end by noting that the somehow more complicated treatment of arbitrary paths expressed by "//" is not due to the generic schema and view definitions that we use; queries involving such paths are difficult to execute in general, since join chains of arbitrary length may be needed to answer them. Structure information is always used to handle such cases, under the form of a dataguide in [11], or DTDs in [13].

Rewriting complex SQL queries The algorithm that we described can handle simple *select-from-where* queries. What we would need is a general algorithm for rewriting of arbitrary SQL queries using arbitrary SQL views. In the case of nested queries, we require an equivalent rewriting of each elementary block and then recompose the rewritings into a nested query similar to the incoming one. This approach may not yield all the solutions, but is guaranteed to find only correct solutions. In [4], algorithms are given for SQL query rewriting using materialized views, in the case when the query and the views are conjunctive SQL queries. Algorithms proposed in [19] solve the rewriting problem in the cases when the query contains aggregation and unions, and the views contain aggregation. We note therefore that solutions exist for a variety of cases, and that identifying relevant tables for answering an XML query is but an instance of the SQL rewriting problem; furthermore, for the particular kind of queries resulting from the translation process, reasoning on the generic schema and on the DTDs provides semantic information that reduces the search space of the rewritings. At the time of this writing, we are not aware of a methodology for rewriting recursive queries, using recursive or non-recursive views, for bag semantics; progress in this area would benefit directly our approach.

8 Conclusion and future work

Using relational storage for XML documents has received considerable attention; several mapping techniques have been proposed, and for some particular mappings, the translation of queries in some XML query language to SQL over the corresponding table has been described. There is no universal policy for choosing "the best" mapping between XML and

relational data; the problem is related to the view selection problem for relational DBMSs, and by nature is likely to be significantly more complex. Therefore, we aimed at freeing the storage designer from the task of the query translation, and we provide a solution to the translation problem regardless of the data mapping.

A novelty of our work is the three-levels framework for *normalizing* queries expressed in the Quilt query language, *translating* them on a virtual generic schema, and *rewriting* these queries, using the relational storage tables as views over the generic schema. At the language level, we analyzed in details the new features from the W3C language algebra and requirements, and we formulated new rewriting and simplification rules for the XML query language; these rules are useful in general for Quilt query processing, whether it is done in relational systems or otherwise, since unnesting eliminates block dependencies and leaves more choices to the optimizer. Also, focusing on the language allowed us to highlight, at the algebraic level, constructs that cannot be dealt with by executing SQL queries and tagging the result. For the translatable subset of the language, we provide translation rules to SQL. Finally, at the SQL query rewriting level, we take advantage of semantic knowledge on the virtual generic schema, as well as on the XML document structure, to adapt and improve an existing SQL rewriting algorithm. Introducing the middle layer in the architecture - the virtual generic schema - is a small price to pay for decoupling automatic query translation and storage design; even more so since the algorithms described here performed well in our implementation (as an example, cummulated time for the three steps was less than 1 second for a query over 25 virtual tables, 3 documents, using 10 views).

As directions for future work, we are interested to see how these techniques carry over to an XML query language that can also update - updates are the next stage in the design of the standard language. Relational systems have a framework of integrity constraints, triggers and such; defining integrity constraints for XML is still an open area of activity, and it is worth considering how far can the relational engines support XML features.

References

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[2] Michael Carey, Daniela Florescu, Zachary Ives, Ying Lu, Jayavel Shanmungasundaram, Eugene Shekita, and Subbu Subramanian. XPERANTO: Publishing object-relational data as XML. In *Proceedings of the International Workshop on the Web and Databases, Houston, USA*, 2000.

[3] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In *Proceedings of the International Workshop on the Web and Databases, Houston, USA*, 2000.

[4] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Taipei, Taiwan, 1995.

[5] Surajit Chaudhuri and Moshe Vardi. Optimizing real conjunctive queries. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 59–70, Washington D.C., 1993.

[6] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query language for XML. In *Proc. of the Int. WWW Conf.*, volume 31(11-16), pages 1155–1169, 1999.

[7] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 431–442, 1999.

[8] Mary Fernandez, Wang-Chiew Tan, and Dan Suciu. Silkroute: Trading between relational and XML. In *Proc. of the Int. WWW Conf.*, May 2000.

[9] S. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. Expressing recursive queries in SQL. ISO/IEC X3H2-96-075r1.

[10] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDMBS. In *IEEE Data Engineering Bulletin*, volume 22(3), pages 27–34, 1999.

[11] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[12] Laks V.S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. SchemaSQL - a language for interoperability in relational multi-database systems. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Mumbai(Bombay), India, 1996.

[13] Jason McHugh and Jennifer Widom. Query optimization for XML. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 315–326, 1999.

[14] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 251–260, Taipei, Taiwan, 1995.

[15] J. Rao, B. Lindsay, G. Lohman, H. Pirahesh, and D. Simmen. Using EELs, a practical approach to outerjoin and antijoin reordering. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2001. To appear.

[16] Jayavel Shanmugasundaram, H. Gang, Kristin Tufte, Chun Zhang, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 302–314, Edinburgh, Scotland, 1999.

[17] Jayavel Shanmungasundaram, Eugene Shekita, Rimon Barr, Michael Carey, Bruce Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as XML documents. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Cairo, Egypt, 2000.

[18] SQL99 ISO standard. ISO/IEC 9075-1999.

[19] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering SQL queries using materialized views. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[20] R. Cattell *et al.* The object database standard - odmg 93. Morgan Kauffmann, 1993.

[21] Dimitri Theodoratos and Timos Sellis. Data warehouse design. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[22] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 367–378, Santiago, Chile, 1994.

[23] XML Information Set. http://www.w3.org/TR/xml-infoset, 2000.

[24] XML Query Algebra. http://www.w3.org/XML/Group/xmlquery/xmlquery-algebra, 2000. Work in progress.

[25] XML Query Requirements. http://www.w3.org/TR/xmlquery-req, 2000. Work in progress.

[26] XML Schema. http://www.w3.org/TR/XML/Schema, 1999.

[27] XPath. http://www.w3.org/TR/xpath, 1999.

[28] XML Query Data Model. http://www.w3.org/TR/query-datamodel, 2000. Work in progress.

[29] Health Level Seven. http://www.hl7.org, 2000.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399